NAME

DBD::Gofer - A stateless-proxy driver for communicating with a remote DBI

SYNOPSIS

```
use DBI;

$original_dsn = "dbi:..."; # your original DBI Data Source Name

$dbh = DBI->connect("dbi:Gofer:transport=$transport;...;dsn=$original_dsn",
$user, $passwd, \%attributes);

... use $dbh as if it was connected to $original_dsn ...
```

The `transport=$transport` part specifies the name of the module to use to transport the requests to the remote DBI. If `$transport` doesn't contain any double colons then it's prefixed with `DBD::Gofer::Transport::`.

The `dsn=$original_dsn` part *must be the last element*of the DSN because ev erything after `dsn=` is assumed to be the DSN that the remote DBI should use.

The `...` represents attributes that influence the operation of the Gofer driver or transport. These are described below or in the documentation of the transport module being used.

DESCRIPTION

DBD::Gofer is a DBI database driver that forwards requests to another DBI driver, usually in a separate process, often on a separate machine. It tries to be as transparent as possible so it appears that you are using the remote driver directly.

DBD::Gofer is very similar to DBD::Proxy. The major difference is that with DBD::Gofer no state is maintained on the remote end. That means every request contains all the information needed to create the required state. (So, for example, every request includes the DSN to connect to.) Each request can be sent to any available server. The server executes the request and returns a single response that includes all the data.

This is very similar to the way http works as a stateless protocol for the web. Each request from your web browser can be handled by a different web server process.

Use Cases

This may seem like pointless overhead but there are situations where this is a very good thing. Let's consider a specific case.

Imagine using DBD::Gofer with an http transport. Your application calls *connect()*, prepare("select * from table where foo=?"), *bind_param()*, and *execute()*. At this point DBD::Gofer builds a request containing all the information about the method calls. It then uses the httpd transport to send that request to an apache web server.

This 'dbi execute' web server executes the request (using DBI::Gofer::Execute and related modules) and builds a response that contains all the rows of data, if the statement returned any, along with all the attributes that describe the results, such as `$sth->{NAME}`. This response is sent back to DBD::Gofer which unpacks it and presents it to the application as if it had executed the statement itself.

Advantages

Okay, but you still don't see the point? Well let's consider what we've gained:

Connection Pooling and Throttling

The 'dbi execute' web server leverages all the functionality of web infrastructure in terms of load balancing, high-availability, firewalls, access management, proxying, caching.

At its most basic level you get a configurable pool of persistent database connections.

Simple Scaling

Got thousands of processes all trying to connect to the database? You can use DBD::Gofer to connect them to your smaller pool of 'dbi execute' web servers instead.

Caching

Client-side caching is as simple as adding `cache=1` to the DSN. This feature alone can be worth using DBD::Gofer for.

Fewer Network Round-trips

DBD::Gofer sends as few requests as possible (dependent on the policy being used).

Thin Clients / Unsupported Platforms

You no longer need drivers for your database on every system. DBD::Gofer is pure perl.

CONSTRAINTS

There are some natural constraints imposed by the DBD::Gofer But not many:

You can't change database handle attributes after *connect()*

You can't change database handle attributes after you've connected. Use the *connect()* call to specify all the attribute settings you want.

This is because it's critical that when a request is complete the database handle is left in the same state it was when first connected.

An exception is made for attributes with names starting `private_`: They can be set after *connect()* but the change is only applied locally.

You can't change statement handle attributes after *prepare()*

You can't change statement handle attributes after prepare.

An exception is made for attributes with names starting `private_`: They can be set after *prepare()* but the change is only applied locally.

You can't use transactions

AutoCommit only. Transactions aren't supported.

(In theory transactions could be supported when using a transport that maintains a connection, like `stream` does. If you're interested in this please get in touch via dbi-dev@perl.org)

You can't call driver-private sth methods

But that's rarely needed anyway.

GENERAL CAVEATS

A few important things to keep in mind when using DBD::Gofer:

Temporary tables, locks, and other per-connection persistent state

You shouldn't expect any per-session state to persist between requests. This includes locks and temporary tables.

Because the server-side may execute your requests via a different database connections, you can't rely on any per-connection persistent state, such as temporary tables, being available from one request to the next.

This is an easy trap to fall into. A good way to check for this is to test your code with a Gofer policy package that sets the `connect_method` policy to 'connect' to force a new connection for each request. The `pedantic` policy does this.

Driver-private Database Handle Attributes

Some driver-private dbh attributes may not be available if the driver has not implemented the *private_attribute_info()* method (added in DBI 1.54).

Driver-private Statement Handle Attributes

Driver-private sth attributes can be set in the *prepare()* call. TODO

Some driver-private sth attributes may not be available if the driver has not implemented the *private_attribute_info()* method (added in DBI 1.54).

Multiple Resultsets
Multiple resultsets are supported only if the driver supports the *more_results()* method (an exception is made for DBD::Sybase).

Statement activity that also updates dbh attributes
Some drivers may update one or more dbh attributes after performing activity on a child sth. For example, DBD::mysql provides `$dbh->{mysql_insertid}` in addition to `$sth->{mysql_insertid}`. Currently mysql_insertid is supported via a hack but a more general mechanism is needed for other drivers to use.

Methods that report an error always return undef
With DBD::Gofer, a method that sets an error always return an undef or empty list. That shouldn't be a problem in practice because the DBI doesn't define any methods that return meaningful values while also reporting an error.

Subclassing only applies to client-side
The RootClass and DbTypeSubclass attributes are not passed to the Gofer server.

CAVEATS FOR SPECIFIC METHODS

last_insert_id
To enable use of last_insert_id you need to indicate to DBD::Gofer that you'd like to use it. You do that my adding a `go_last_insert_id_args` attribute to the *do()* or *prepare()* method calls. For example:

```
$dbh->do($sql, { go_last_insert_id_args => [...] });
```

or

```
$sth = $dbh->prepare($sql, { go_last_insert_id_args => [...] });
```

The array reference should contains the args that you want passed to the *last_insert_id()* method.

execute_for_fetch
The array methods*bind_p aram_array()* and *execute_array()* are supported. When *execute_array()* is called the data is serialized and executed in a single round-trip to the Gofer server. This makes it very fast, but requires enough memory to store all the serialized data.

The *execute_for_fetch()* method currently isn't optimised, it uses the DBI fallback behaviour of executing each tuple individually. (It could be implemented as a wrapper for *execute_array()* - patches welcome.)

TRANSPORTS

DBD::Gofer doesn't concern itself with transporting requests and responses to and fro. For that it uses special Gofer transport modules.

Gofer transport modules usually come in pairs: one for the 'client' DBD::Gofer driver to use and one for the remote 'server' end. They have very similar names:

```
DBD::Gofer::Transport::<foo>
DBI::Gofer::Transport::<foo>
```

Sometimes the transports on the DBD and DBI sides may have different names. For example DBD::Gofer::Transport::http is typically used with DBI::Gofer::Transport::mod_perl (DBD::Gofer::Transport::http and DBI::Gofer::Transport::mod_perl modules are part of the GoferTransport-http distribution).

Bundled Transports
Several transport modules are provided with DBD::Gofer:

null

The null transport is the simplest of them all. It doesn't actually transport the request anywhere. It just serializes (freezes) the request into a string, then thaws it back into a data structure before passing it to DBI::Gofer::Execute to execute. The same freeze and thaw is applied to the results.

The null transport is the best way to test if your application will work with Gofer. Just set the DBI_AUTOPROXY environment variable to `dbi:Gofer:transport=null;policy=pedantic` (see "Using DBI_AUTOPROXY" below) and run your application, or ideally its test suite, as usual.

It doesn't take any parameters.

pipeone

The pipeone transport launches a subprocess for each request. It passes in the request and reads the response.

The fact that a new subprocess is started for each request ensures that the server side is truly stateless. While this does make the transport *very* slow, it is useful as a way to test that your application doesn't depend on per-connection state, such as temporary tables, persisting between requests.

It's also useful both as a proof of concept and as a base class for the stream driver.

stream

The stream driver also launches a subprocess and writes requests and reads responses, like the pipeone transport. In this case, however, the subprocess is expected to handle more that one request. (Though it will be automatically restarted if it exits.)

This is the first transport that is truly useful because it can launch the subprocess on a remote machine using `ssh`. This means you can now use DBD::Gofer to easily access any databases that's accessible from any system you can login to. You also get all the benefits of ssh, including encryption and optional compression.

See "Using DBI_AUTOPROXY" below for an example.

Other Transports

Implementing a Gofer transport is *very* simple, and more transports are very welcome. Just take a look at any existing transports that are similar to your needs.

http

See the GoferTransport-http distribution on CPAN: http://search.cpan.org/dist/GoferTransport-http/

Gearman

I know Ask Bjørn Hansen has implemented a transport for the `gearman` distributed job system, though it's not on CPAN at the time of writing this.

CONNECTING

Simply prefix your existing DSN with `dbi:Gofer:transport=$transport;dsn=` where `$transport` is the name of the Gofer transport you want to use (see "TRANSPORTS"). The `transport` and `dsn` attributes must be specified and the `dsn` attributes must be last.

Other attributes can be specified in the DSN to configure DBD::Gofer and/or the Gofer transport module being used. The main attributes after `transport`, are `url` and `policy`. These and other attributes are described below.

Using DBI_AUTOPROXY

The simplest way to try out DBD::Gofer is to set the DBI_AUTOPROXY environment variable. In this case you don't include the `dsn=` part. For example:

```
export DBI_AUTOPROXY="dbi:Gofer:transport=null"
```

or, for a more useful example, try:

```
export DBI_AUTOPROXY="dbi:Gofer:transport=stream;url=ssh:user@example.com"
```

Connection Attributes

These attributes can be specified in the DSN. They can also be passed in the %attr parameter of the DBI connect method by adding a `go_` prefix to the name.

transport

Specifies the Gofer transport class to use. Required. See "TRANSPORTS" above.

If the value does not include `::` then `DBD::Gofer::Transport::` is prefixed.

The transport object can be accessed via `$h->{go_transport}`.

dsn

Specifies the DSN for the remote side to connect to. Required, and must be last.

url

Used to tell the transport where to connect to. The exact form of the value depends on the transport used.

policy

Specifies the policy to use. See "CONFIGURING BEHAVIOUR POLICY".

If the value does not include `::` then `DBD::Gofer::Policy` is prefixed.

The policy object can be accessed via `$h->{go_policy}`.

timeout

Specifies a timeout, in seconds, to use when waiting for responses from the server side.

retry_limit

Specifies the number of times a failed request will be retried. Default is 0.

retry_hook

Specifies a code reference to be called to decide if a failed request should be retried. The code reference is called like this:

```
$transport = $h->{go_transport};
$retry = $transport->go_retry_hook->($request, $response, $transport);
```

If it returns true then the request will be retried, up to the `retry_limit`. If it returns a false but defined value then the request will not be retried. If it returns undef then the default behaviour will be used, as if `retry_hook` had not been specified.

The default behaviour is to retry requests where `$request->`is_idempotent is true, or the error message matches `/induced by DBI_GOFER_RANDOM/`.

cache

Specifies that client-side caching should be performed. The value is the name of a cache class to use.

Any class implementing get($key) and set($key, `$value`) methods can be used. That includes a great many powerful caching classes on CPAN, including the Cache and Cache::Cache distributions.

You can use `cache=1`" is a shortcut for "`cache=DBI::Util::CacheMemory`. See DBI::Util::CacheMemory for a description of this simple fast default cache.

The cache object can be accessed via `$h->`go_cache. For example:

```
$dbh->go_cache->clear; # free up memory being used by the cache
```

The cache keys are the frozen (serialized) requests, and the values are the frozen responses.

The default behaviour is to only use the cache for requests where `$request->`is_idempotent is true (i.e., the dbh has the ReadOnly attribute set or the SQL statement is obviously a SELECT without a FOR UPDATE clause.)

For even more control you can use the `go_cache` attribute to pass in an instantiated cache object. Individual methods, including *prepare()*, can also specify alternative caches via the `go_cache`

attribute. For example, to specify no caching for a particular query, you could use

```
$sth = $dbh->prepare( $sql, { go_cache => 0 } );
```

This can be used to implement different caching policies for different statements.

It's interesting to note that DBD::Gofer can be used to add client-side caching to any (gofer compatible) application, with no code changes and no need for a gofer server. Just set the DBI_AUTOPROXY environment variable like this:

```
DBI_AUTOPROXY='dbi:Gofer:transport=null;cache=1'
```

CONFIGURING BEHAVIOUR POLICY

DBD::Gofer supports a 'policy' mechanism that allows you to fine-tune the number of round-trips to the Gofer server. The policies are grouped into classes (which may be subclassed) and referenced by the name of the class.

The DBD::Gofer::Policy::Base class is the base class for all the policy packages and describes all the available policies.

Three policy packages are supplied with DBD::Gofer:

DBD::Gofer::Policy::pedantic is most 'transparent' but slowest because it makes more round-trips to the Gofer server.

DBD::Gofer::Policy::classic is a reasonable compromise - it's the default policy.

DBD::Gofer::Policy::rush is fastest, but may require code changes in your applications.

Generally the default `classic` policy is fine. When first testing an existing application with Gofer it is a good idea to start with the `pedantic` policy first and then switch to `classic` or a custom policy, for final testing.

AUTHOR

Tim Bunce, <http://www.tim.bunce.name>

LICENCE AND COPYRIGHT

ACKNOWLEDGEMENTS

The development of DBD::Gofer and related modules was sponsored by Shopzilla.com (<http://Shopzilla.com>), where I currently work.

SEE ALSO

DBI::Gofer::Request, DBI::Gofer::Response, DBI::Gofer::Execute.

DBI::Gofer::Transport::Base, DBD::Gofer::Policy::Base.

DBI

Caveats for specific drivers

This section aims to record issues to be aware of when using Gofer with specific drivers. It usually only documents issues that are not natural consequences of the limitations of the Gofer approach - as documented above.

TODO

This is just a random brain dump... (There's more in the source of the Changes file, not the pod)

Document policy mechanism

Add mechanism for transports to list config params and for Gofer to apply any that match (and warn if any left over?)

Driver-private sth attributes - set via *prepare()* - change DBI spec

add hooks into transport base class for checking & updating a result set cache ie via a standard

cache interface such as:
http://search.cpan.org/~robm/Cache-FastMmap/FastMmap.pm
http://search.cpan.org/~bradfitz/Cache-Memcached/lib/Cache/Memcached.pm
http://search.cpan.org/~dclinton/Cache-Cache/
http://search.cpan.org/~cleishman/Cache/ Also caching instructions could be passed through the httpd transport layer in such a way that appropriate http cache headers are added to the results so that web caches (squid etc) could be used to implement the caching. (MUST require the use of GET rather than POST requests.)

Rework handling of installed_methods to not piggyback on dbh_attributes?

Perhaps support transactions for transports where it's possible (ie null and stream)? Would make stream transport (ie ssh) more useful to more people.

Make sth_result_attr more like dbh_attributes (using '*' etc)

Add `@val` = FETCH_many(@names) to DBI in C and use in Gofer/Execute?

Implement _new_sth in C.